OCaml Exceptions

```
exception My_exception of int
let f n =  
  if n > 0 then  
    raise (My_exception n)  
  else  
    raise (Failure "foo")  
let bar n =  
  try  
    f n  
  with  
    My_exception n ->  
      Printf.printf "Caught %d\n" n  
  | Failure s ->  
      Printf.printf "Caught %s\n" s
```

OCaml Exceptions (cont.)

- Exceptions are declared with `exception`
  - They may appear in the signature as well
- Exceptions may take arguments
  - Just like type constructors
  - May also be nullary
- Catch exceptions with `try...with...`
  - Pattern-matching can be used in with
  - If an exception is uncaught
    - Current function exits immediately
    - Control transfers up the call chain
    - Until the exception is caught, or reaches the top level

Modules

- So far, most everything we’ve defined
  - Has been at the “top-level” of OCaml
  - This is not good software engineering practice
- A better idea
  - Use modules to group together associated
    - Types, functions, and data
  - Avoid polluting the top-level with unnecessary stuff
- For lots of sample modules
  - See the OCaml standard library
Modularity and Abstraction

- Another reason for creating a module
  - So we can hide details
  - Example
    - Build a binary tree module
    - Hide exact representation of binary trees
  - This is also good software engineering practice
    - Prevents clients from relying on details that may change
    - Hides unimportant information
    - Promotes local understanding (clients can’t inject arbitrary data structures, only ones our functions create)

Creating a Module in OCaml

```ocaml
module Shapes =
  struct
    type shape =
      Rect of float * float (* wid*len *)
    | Circle of float (* radius *)
    let area = function
      Rect (w, l) -> w *. l
    | Circle r -> r *. r *. 3.14
    let unit_circle = Circle 1.0
  end;
```

Creating a Module in OCaml (cont.)

```ocaml
module Shapes =
  struct
    type shape = _
    let area = _
    let unit_circle = _
  end;
  unit_circle; (* not defined *)
  Shapes.unit_circle;
  Shapes.area (Shapes.Rect (3.0, 4.0));;
  open Shapes; (* import names into curr scope *)
  unit_circle; (* now defined *)
```

Module Signatures

- Entry in signature
- Supply function types

```ocaml
module type FOO =
  sig
    val add : int -> int -> int
  end;
module Foo : FOO =
  struct
    let add x y = x + y
    let mult x y = x * y
  end;
  Foo.add 3 4;; (* OK *)
  Foo.mult 3 4;; (* not accessible *)
```

Module Signatures (cont.)

- Convention
  - Signatures to be all capital letters
  - This isn’t a strict requirement, though
- Items can be omitted from a module signature
  - This provides the ability to hide values
- The default signature for a module hides nothing
  - You’ll notice this is what OCaml gives you if you just type in a module with no signature at the top-level

Modularity

- Definition
  - Extent to which a computer program is composed of separate parts
  - Higher degree of modularity is better
- Modular programming
  - Programming techniques that increase modularity
    - Interface vs. implementation
- Modular programming languages
  - Explicit support for modules
    - Ada, Fortran, ML, Modula-2, Python, Ruby, OCaml
Abstract Types in Signatures

```ocaml
module type SHAPES =
  sig
    type shape
  end

module Shapes : SHAPES =
  let make_circle r = Circle r
  let make_rect x y = Rect (x, y)
```

Now definition of `shape` is hidden.

.ml and .mli files

- Put the signature in a `foo.ml` file, the struct in a `foo.mli` file
  - Use the same names
  - Omit the `sig...end` and `struct...end` parts
  - The OCaml compiler will make a Foo module from these

Example – OCaml Module Signatures

```ocaml
module type shape =
  sig
    val area : shape -> float
    val unit_circle : shape
    val make_circle : float -> shape
    val make_rect : float -> float -> shape
  end

module Shapes : shape =
  let make_circle r = Circle r
  let make_rect x y = Rect (x, y)
```

### Functors

- Modules can take other modules as arguments
  - Such a module is called a functor
  - You're mostly on your own if you want to use these

#### Example: `Set` in standard library

```ocaml
module type OrderedType = sig
  type t
  val compare : t -> t -> int
end

module Make(OrderedType) =
  struct ...
  end

module StringSet = Make(String);
(* works because String has type t, implements compare *)
```

Module in Java

- Java `classes` are like modules
  - Provides implementations for a group of functions
  - But classes can also
    - Instantiate objects
    - Inherit attributes from other classes

- Java `interfaces` are like module signatures
  - Defines a group of functions that may be used
  - Implementation is hidden
Module in C

- .c files are like modules
  - Provides implementations for a group of functions

- .h files are like module signatures
  - Defines a group of functions that may be used
  - Implementation is hidden

- Usage is not enforced by C language
  - Can put C code in .h file

Module in Ruby

- Ruby explicitly supports modules
  - Modules defined by module ... end
  - Modules cannot
    - Instantiate objects
    - Derive subclasses

```
puts Math.sqrt(4)  # 2
puts Math::PI      # 3.1416
include Math       # open Math
puts Sqrt(4)       # 2
puts PI            # 3.1416
```

So Far, Only Functional Programming

- We haven’t given you any way so far to change something in memory
  - All you can do is create new values from old

- This actually makes programming easier!
  - Don’t care whether data is shared in memory
    - Aliasing is irrelevant
  - Provides strong support for compositional reasoning and abstraction
    - Example: Calling a function f with argument x always produces the same result
  - But could take (much) more memory & time to execute

Imperative OCaml

- There are three basic operations on memory
  1) ref : 'a -> 'a ref
     - Allocate an updatable reference
  2) ! : 'a ref -> 'a
     - Read the value stored in reference
  3) := : 'a ref -> 'a -> unit
     - Write to a reference

```
let x = ref 3 (* x : int ref *)
let y = !x
x := 4
```

Comparison to L- and R-values

- Recall that in C/C++/Java, there’s a strong distinction between l- and r-values
  - An r-value refers to just a value, like an integer
  - An l-value refers to a location that can be written

- A variable’s meaning depends on where it appears
  - On the right-hand side, it’s an r-value, and it refers to the contents of the variable
  - On the left-hand side of an assignment, it’s an l-value, and it refers to the location the variable is stored in

L-Values and R-Values in C (cont.)

- Notice that x, y, 3 all have the same type: int
Comparison to OCaml

```ocaml
int x; let x = ref 0;
int y; let y = ref 0;
x = 3; x := 3;; (* x : int *)
y = x; y := (!x);;
3 = x; 3 := x;; (* 3 : int; error *)
```

- In OCaml, an updatable location and the contents of the location have different types
  - The location has a ref type

Capturing a ref in a Closure

- We can use refs to make things like counters that produce a fresh number “everywhere”

```ocaml
let next =
  let count = ref 0 in
  function () ->
  let temp = !count in
  count := (!count) + 1;
  temp;;
# next ();;
- : int = 0
# next ();;
- : int = 1
```

Semicolon Revisited; Side Effects

- Now that we can update memory, we have a real use for ; and () : unit
  - e1; e2 means evaluate e1, throw away the result, and then evaluate e2, and return the value of e2
  - () means “no interesting result here”
  - It’s only interesting to throw away values or use ()
    - If computation does something besides return a result
  - A side effect is a visible state change
    - Modifying memory
    - Printing to output
    - Writing to disk

Grouping with begin...end

- If you’re not sure about the scoping rules, use begin...end to group together statements with semicolons

```ocaml
let x = ref 0
let f () =
  begin
    print_string "hello";
    x := (!x) + 1
  end
```

The Trade-Off of Side Effects

- Side effects are absolutely necessary
  - That’s usually why we run software!
  - We want something to happen that we can observe

- But...they also make reasoning harder
  - Order of evaluation now matters
  - Calling the same function in different places may produce different results
  - Aliasing is an issue
    - If we call a function with refs r1 and r2, it might do strange things if r1 and r2 are aliased

OCaml Language Choices

- Implicit or explicit declarations?
  - Explicit – variables must be introduced with let before use
  - But you don’t need to specify type of variable

- Static or dynamic types?
  - Static – but without type declarations
  - OCaml does type inference to figure out types for you
    - Advantage – less work to write programs
    - Disadvantages – easier to make mistakes, harder to find errors
Higher-Order Functions in C

- C supports function pointers

```c
typedef int (*int_func)(int);
void app(int_func f, int *a, int n) {
    for (int i = 0; i < n; i++)
        a[i] = f(a[i]);
}
int add_one(int x) { return x + 1; }
int main() {
    int a[] = {5, 6, 7};
    app(add_one, a, 3);
}
```

Higher-Order Functions in C (cont.)

- C does not support closures
  - Since no nested functions allowed
  - Unbound symbols always in global scope

```c
int y = 1;
void app(int (*f)(int), n) {
    return f(n);
}
int add_y(int x) {
    return x + y;
}
int main() {
    app(add_y, 2);
}
```

Higher-Order Functions in C (cont.)

- Cannot access non-local variables in C
- OCaml code

```ocaml
let add x y = x + y
```

- Equivalent code in C is illegal

```c
int (*add(int x))(int) {
    return add_y;
}
int add_y(int y) {
    return x + y; // x undefined
}
```

Higher-Order Functions in Ruby

- Ruby supports higher-order functions
  - Use `yield` within method to call code block argument

```ruby
def my_collect(a)
    b = Array.new(a.length)
    0.upto(a.length-1) {
        |i|
        b[i] = yield a[i]
    }
    return b
end
b = my_collect([5, 6, 7]) ( |x| x+1 )
```

Higher-Order Functions in Ruby (cont.)

- Ruby supports closures
  - Code blocks can access non-local variables
  - Binding determined by lexical scoping

```ruby
def twice
    yield
    yield
end
x = 1
twice {x += 1}
puts x # 3
def twice
    x = 0 #dynamic
    yield
    yield
end
x = 1 #lexical
twice {x += 1}
puts x # 3 not 1
```
Higher-Order Functions in Ruby (cont.)

- Ruby code blocks are actual variables
  ```ruby
def twice # implicit block
  yield # invoked with yield
  yield end
  twice { x += 1 } # same as x += 2
  ```

- Code blocks may be saved
  ```ruby
def quad (&block) # explicit block
  twice (c) # used as argument
twice (c) end
  quad { x += 1 } # same as x += 4
  ```

Higher-Order Functions in Ruby (cont.)

- Ruby supports creating closures directly
  ```ruby
  c1 = Proc.new { x+=1 }
c2 = proc { x+=1 }
c3 = lambda { x+=1 }
def foo
  x+=1
  end
c4 = method { :foo }
c.call # x+=1
  ```

An object in Java or C++ is kind of like a closure

- It has some data (like an environment)
- Along with some methods (i.e., function code)
- So objects can be used to simulate closures

Higher-Order Functions in Java/C++

- So is an anonymous Java inner class
  - Inner class methods can access fields of outer class

- Back in CMSC 132 (OOP II)
  - We studied how to implement some functional patterns in OO languages